Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matija Rezar

Vzporedni poboti

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: dr. Andrej Brodnik

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pomembno področje razvoja algoritmov in podatkovnih struktur so vzporedni ter porazdeljeni algoritmi in podatkovne strukture. Z njimi dosežemo boljšo časovno zahtevnost vendar včasih za ceno povečanja opravljenega dela. Pogosto se zatakne pri izvedbi tako zasnovanih rešitev v praksi, saj so orodja in okolja relativno okorna ali zakrivajo implementacijo, kar ima za posledico neučinkovite izvedbe.

V diplomski nalogi zasnujte algoritem, ki bo izkoriščal možnost tesno sklopljenega vzporednega izvajanja (npr. PRAM CREW) in šibko sklopljenega porazdeljenega izvajanja (npr. gruča). Kot primer naj algoritem rešuje problem pobotov med strankami. Pri izvedbi uporabite splošno dostopne tehnologije ter ocenite njihovo primernost.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matija Rezar, z vpisno številko **63100342**, sem avtor diplomskega dela z naslovom:

Vzporedni poboti

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. septembra 2014 Podpis avtorja:

Zahvaljujem se staršem, ki me prenašajo in mentorju dr. Andreju Brodniku, ki me je silil dlje kot sem mislil, da sem sposoben.

Bralcu,
hvala za tvoj čas.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DAG	*directed acyclic graph*	usmerjen acikličen graf
SCC	*strongly connected component*	krepko povezana komponenta
NIF	*native implemented function*	funkcija v jeziku sistema
BFS	*breadth-first search*	iskanje v širino
DFS	*depth-first search*	iskanje v globino
OTP	*open telecom platform*	odprta platforma za telekomunikacije
PRAM	*parallel random access memory*	pomnilnik z vzporednim naključnim dostopom
CREW	*concurrent read - exclusive write*	sočasno branje - posamično pisanje

Povzetek

V jeziku Erlang smo s pomočjo knjižnice napisane v jeziku C, ki uporablja vmesnik OpenMP implementirali algoritem, ki uporablja tako vzporedno kot porazdeljeno računanje za iskanje ciklov v grafu, ki predstavlja dolžnike. Te cikle nato uporabimo za izvedbo pobotov med dolžniki.

Cilj naloge je bil ugotoviti, ali je Erlang primeren za implementacijo razdeljevalnika za porazdeljen sistem. Po testiranju na naključno zgrajenih grafih majhnega sveta smo prišli do zaključka, da Erlang v čisti obliki za tako nalogo ni primeren in je potrebno poiskati druge rešitve.

Ključne besede: graf, cikel, "usmerjen graf", "vzporedno programiranje", "porazdeljeno programiranje", "odkrivanje ciklov".

Abstract

Using Erlang and a library written in C using OpenMP we implemented an algorithm that utilizes both parallel and distributed computing to find cycles in a graph which represents debtors. These cycles are then used to perform debt reconciliation between debtors.

In the thesis we attempt to establish whether Erlang is suitable for the implementation of a work distribution component in a distributed system. After testing on random generated small-world graphs we conclude that Erlang in it's pure form is not appropriate for that task.

Keywords: graph, cycle, "directed graph", "parallel programming", "distributed programming", "cycle detection".

Poglavje 1

Uvod

V nalogi si bomo ogledali implementacijo porazdeljenega sistema za reševanje pobotov dolgov med uporabniki na osnovi Erlanga z implementacijo iskanja krepko povezanih komponent in ciklov z vzporednim algoritmom implementiranim v jeziku C.

Gre za problem izvajanja pobotov dolgov med uporabniki socialne aplikacije, ki za izračun pobotov uporablja porazdeljen sistem računalnikov z večjedrnimi procesorji. Tak sistem bi lahko uporabljal navidezne računalnike v oblaku, npr. Amazon EC2, ali gručo sestavljeno iz računalnikov osnovanih na potrošniški strojni opremi.

1.1 Opredelitev problema

Problem modeliramo kot usmerjen graf, kjer vsako vozlišče predstavlja uporabnika, ki je dolžnik in/ali upnik. Povezave v grafu predstavljajo dolg izvornega vozlišča povezave do ponornega vozlišča povezave.

Če dolgovi tvorijo cikel lahko dolžnik z najmanjšim dolgom svoj dolg poplača, ta denar pa nato potuje po ciklu in se mu na koncu vrne. Lažji način je, da vsi upniki v ciklu odpišejo del dolga enak najmanjšemu dolgu v ciklu. Tako se teža vseh povezav v ciklu zmanjša za težo najmanjše povezave in vse povezave s težo 0 se odstranijo iz grafa. To ponavljamo dokler lahko,

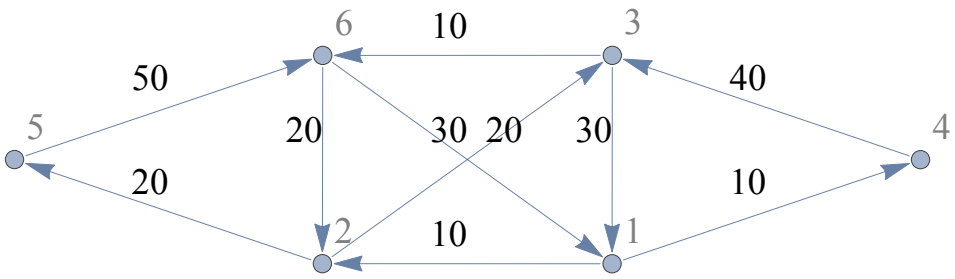
se pravi dokler v grafu obstajajo cikli.

Končni cilj algoritma je proizvesti usmerjen acikličen graf - DAG, kjer so ,,najmanjše“[1] povezave odstranjene, teže ostalih povezav pa so zmanjšane. To pomeni, da v grafu ne obstaja več nobena skupina dolgov, ki bi jih lahko med seboj pobotali.

Ker reševanje problemov na grafih hitro postane časovno zahtevno želimo uporabiti način, ki za reševanje izrablja večjedrne procesorje. Ker pa se tudi bližamo koncu rasti po Moorovem zakonu [17], hkrati pa raste popularnost računalništva v oblaku, je pametno zasnovati algoritem, ki za delovanje izrablja porazdeljen sistem.

1.2 Primer

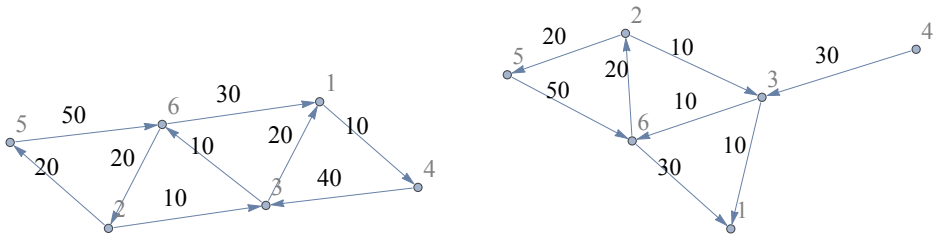
Na sliki 1.1 je začetno stanje grafa. Vozlišča predstavljajo dolžnike, povezave pa dolgove med njimi. Teže povezav predstavljajo velikosti dolgov. Iskanje bomo začeli v vozlišču 1.



Slika 1.1: Začetno stanje.

Najprej najdemo cikel $1 \to 2 \to 3$. Najmanjša povezava je $1 \to 2$ s težo 10, zato jo odstranimo in zmanjšamo ostali povezavi za 10. Dobimo graf na sliki 1.2a.

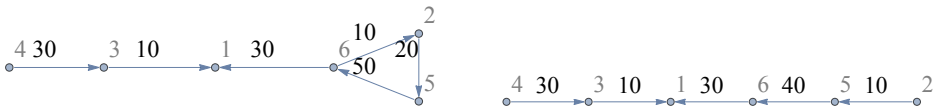
[1]Povezave niso nujno najmanjše v celotnem grafu; so le najmanjše v ciklu, kjer se pojavijo.



(a) Prvi cikel (b) Drugi cikel

Slika 1.2: Odstranjena prvi in drugi cikel.

Nato obdelamo še cikla $1 \to 4 \to 3$ in $2 \to 3 \to 6$. Pri slednjem si mesto najmanjše povezave delita dve povezavi zato odstranimo obe.



(a) Tretji cikel (b) Četrti cikel

Slika 1.3: Odstranjena tretji in četrti cikel.

Odstranimo še cikel $2 \to 5 \to 6$ in dobimo DAG na sliki 1.3b.

Sedaj moramo najti način kako delo porazdeliti med več računalnikov in več procesorjev.

1.3 Struktura naloge

V uvodu smo predstavili definicijo naloge in kratko demonstracijo reševanja. V poglavju 2 si bomo ogledali kakšne so rešitve, ki za reševanje naloge že obstajajo. Poglavje 3 vsebuje podrobnejši opis izdelane rešitve in njenih sestavnih delov. Predzadnje poglavje opisuje rezultate izvajanja programa. V zaključku pa so napisane sklepne ugotovitve.

Poglavje 2

Pregled obstoječih rešitev

2.1 Zaporedne rešitve

Iskanje ciklov Najbolj osnoven algoritem za iskanje ciklov temelji na iskanju v globino, kjer v globino iščemo vozlišče, ki smo ga že prej obiskali [6].

Iskanje krepko povezanih komponent Eden najbolj preprostih algoritmov na tem področju je algoritem Kosaraju-Sharir [3, 6, 15], ki je asimptotično optimalen. Bolj učinkovit pa je Tarjanov algoritem za iskanje krepko povezanih komponent [16].

Algoritem Kosaraju-Sharir (Algoritem 1) deluje tako, da si izmed neobiskanih vozlišč izbere eno, iz njega pregleduje graf v globino in ob vračanju daje obiskana vozlišča na sklad. To ponavlja dokler niso vsa vozlišča na skladu. Nato povezave v grafu obrne. Dokler sklad ni prazen jemlje po eno vozlišče s sklada in iz njega ponovi pregled. Vozlišča, ki jih najde so v krepko povezani komponenti skupaj z začetnim vozliščem in jih odstrani iz sklada in grafa.

Tarjanov algoritem (Algoritem 2) je podoben algoritmu Kodaraju-Sharir, vendar že v prvem obhodu gradi podatke o povratnih povezavah in proizvede krepko povezane komponente že v enem pregledu.

Algoritem 1: Algoritem Kosaraju-Sharir.

```
while Niso obiskana vsa vozlišča v grafu do
    /* Išči v globino do že obiskanih vozlišč in po
       rekurzivnem klicu dodaj vozlišče na sklad S.      */
end
/* Obrni vse povezave v grafu.                           */
while S ni prazen do
    /* Vzemi vozlišče v s sklada S                       */
    /* Išči v globino iz v.  Vsa vozlišča na katera naletiš
       so v SCC z v.                                     */
    /* Vozlišča v SCC odstrani z grafa in sklada.        */
end
```

2.2 Porazdeljeno računanje

Na tem področju najdemo obilico rešitev. Najbolj osnovna je tehnologija MPI [9], ki omogoča drobnozrnato upravljanje s porazdeljenim sistemom, ampak zahteva predhodno definicijo podatkov in natančno upravljanje z njimi, prinese pa visoko zmogljivost. Bolj kompleksna je rešitev BOINC [5], ki na višjem nivoju razporeja delo in prevzema rezultate ter beleži velikost prispevka vsakega računalnika. Tretja, malo bolj nenavadna rešitev pa je uporaba Erlanga in njegovih vgrajenih mehanizmov za razpošiljanje sporočil in porazdeljeno delovanje.

Standard MPI se uporablja predvsem v visoko zmogljivih gručah, kjer so računalniki povezani s hitrimi povezavami. Namenjen je hitri in učinkoviti komunikaciji znotraj visokozmogljivostnih sistemov.

BOINC je dozorel kot del projekta SETI@home, ki se ukvarja z analizo radijskih signalov iz vesolja. Omogoča, da uporabniki po celem svetu dajo v uporabo svoj računalnik, s svojimi prispevki pa med seboj tekmujejo.

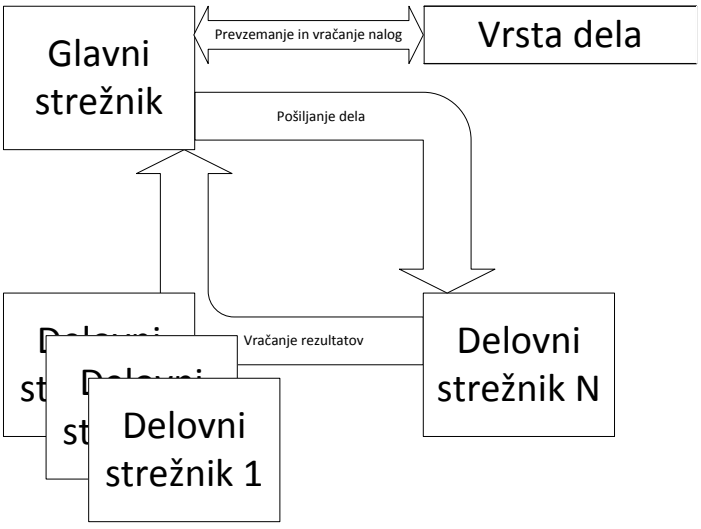
Erlang je funkcijski jezik, zasnovan za programiranje telefonskih central kjer se v eni od implementacij ponaša z zanesljivostjo ,,devetih devetic“. Med

Algoritem 2: Tarjanov algoritem za iskanje krepko povezanih komponent [2, 16].

```
index = 0;
S = [];
foreach v ∈ V do
    if v.index = undefined then
        strongconnect(v);
    end
end
function strongconnect(v)
    v.index = index;
    v.lowlink = index;
    index = index + 1;
    S.push(v);
    foreach (v, w) ∈ E do
        if w.index = undefined then
            strongconnect(w);
            v.lowlink = min(v.lowlink, w.lowlink);
        end
        else if w ∈ S then
            v.lowlink = min(v.lowlink, w.index);
        end
    end
    if v.lowlink = v.index then
        /* Začni novo SCC.                    */
        repeat
            w = S.pop();
            /* Dodaj w v SCC.                 */
        until w = v;
        /* Izpiši SCC.                        */
    end
end
```

njegove pomembnejše lastnosti sodijo usmerjenost k množici lahkih procesov in vgrajeni konstrukti za medprocesno komunikacijo. Najbolj popularna implementacija Erlang/OTP vsebuje obnašanja[1] posebej namenjena nadzoru procesov, hitri implementaciji strežnikov in končnih avtomatov ter spremljanju dogodkov.

Med tem ko BOINC že vsebuje mehanizme za razdeljevanje dela, pa je te potrebno pri MPIju in Erlangu posebej implementirati. Sledili bomo modelu opisanem na sliki 2.1. Glavni strežnik hrani vrsto dela. Ko so na voljo delovni strežniki iz vrste vzame delo in ga pošlje delovnemu strežniku. Ta nalogo obdela in vrne rezultate. Del rezultatov so lahko tudi podatki o nadaljnjem delu, ki ga glavni strežnik ponovno da v vrsto. Implementacija takega sistema s pomočjo MPIja ali Erlanga nam omogoča, da se znebimo nepotrebnih delov BOINCa in jo prilagodimo točno določenemu problemu.



Slika 2.1: Diagram delovanja porazdeljenega sistema. Glavni strežnik iz vrste jemlje delo in ga pošilja delovnim strežnikom. Ti vračajo rezultate, na podlagi katerih glavni strežnik lahko da novo delo v vrsto.

[1]Obnašanja so podobna Javanskim vmesnikom in definirajo nabor funkcij, ki jih modul implementira.

Vse tri rešitve omogočajo implementacijo računsko zahtevnejših delov v C [1, 8], kar pomeni, da je izbira odvisna od lastnosti problema.

2.3 Vzporedno računanje

Tehnologija Na vozliščih porazdeljenega sistema bomo podatke obdelovali vzporedno. Za ta del sta najbolj očitna kandidata OpenMP [7] in OpenCL [10]. Prvi ponuja preprosto prilagoditev obstoječe kode napisane v C za vzporedno delovanje, drugi pa omogoča razširitev na nehomogene sisteme, vendar zahteva kodo napisano v svojem jeziku.

Pretvorba C kode v vzporedno poteka z vstavljanjem OpenMP predprocesorskih ukazov v obstoječo kodo. S temi ukazi označimo dele kode za katere želimo, da se izvajajo vzporedno, pri zankah pa lahko vsaki niti dodelimo del podatkov za obdelavo.

Listing 2.1: Primer kode, ki uporablja OpenMP.

```
#include <stdio.h>
#include <omp.h>

int main(void){

        int* a = malloc(sizeof(int)*10);

        /* Ta del se izvaja zaporedno v eni niti. */
        for(int i = 0; i < 10; i++){
                a[i] = i;
        }

        #pragma omp parallel
        {
                /* Ta del ponovi vsaka nit. */
```

```
            printf("Hello world!");

            /* Ta del si niti razdelijo tako, */
            /* da vsaka obdela svoj del polja. */
            #pragma omp for
            for(int i = 0; i < 10 i++){
                    a[i] = a[i] * a[i];
            }
        }
        return 0;
}
```

OpenCL lahko isto kodo uporablja za računanje na različnih procesnih enotah od običajnih centralnih procesnih enot, grafičnih kartic pa do FPGA čipov. S pomočjo BOINCa ga projekt Einstein@Home uporablja za razpoznavanje zunajzemeljskih radijskih signalov. BOINC ima kot prostovoljen projekt še posebej heterogeno strojno opremo, zato tam OpenCL pride do izraza.

Algoritem Iskali bomo krepko povezane komponente v grafu - SCC[2], za kar je bil že narejen vzporeden algoritem FW-BW (algoritem 3) [12], ki pa uporablja iskanje v širino. Na srečo za to obstaja optimalen algoritem [11], katerega časovna zahtevnost je v najslabšem primeru $O(|V| + |E|)$[3].

[2]Krepko povezana komponenta je podgraf v katerem je vsako vozlišče dosegljivo iz vsakega drugega vozlišča, torej obstaja pot med poljubnim parom vozlišč.

[3]Dokaz v poglavju 3.

Algoritem 3: Pseudokoda za algoritem FW-BW

```
function FW-BW(G(V, E))
    v = poljubno vozlišče iz V;
    /* Iskanje v globino označuje vsa obiskana vozlišča z F
       -- vozlišča dosegljiva iz v.                          */
    ParDFS(v);
    /* Obrni vse povezave v E.                               */
    /* Iskanje v globino označuje vsa obiskana vozlišča z B
       -- vozlišče v je dosegljivo iz teh vozlišč.          */
    ParDFS(v);
    SCC = FW = BW = R = ∅;
    foreach v ∈ V do
        if v.F && v.B then
            SCC.insert(v);
        end
        else if v.F && !v.B then
            FW.insert(v);
        end
        else if !v.F && v.B then
            BW.insert(v);
        end
        else if !v.F && !v.B then
            R.insert(v);
        end
    end
    /* E' so povezave znotraj vozlišč v podgrafu.            */
    return {SCC} ∪ FW-BW(G(FW, E'_1)) ∪ FW-BW(G(BW, E'_2)) ∪
    FW-BW(G(R, E'_3))
end
```

Poglavje 3

Opis rešitve

Končna rešitev je sistem, ki s pomočjo implementacije `gen_server` iz ogrodja Erlang/OTP razpošilja delo delovnim strežnikom, ti pa podatke obdelajo s pomočjo v C implementirane NIF[1] funkcije.

3.1 Algoritem

3.1.1 Uporaba krepko povezanih komponent

Pri iskanju cikla se bomo zanašali na lastnosti krepko povezanih komponent v usmerjenem grafu.

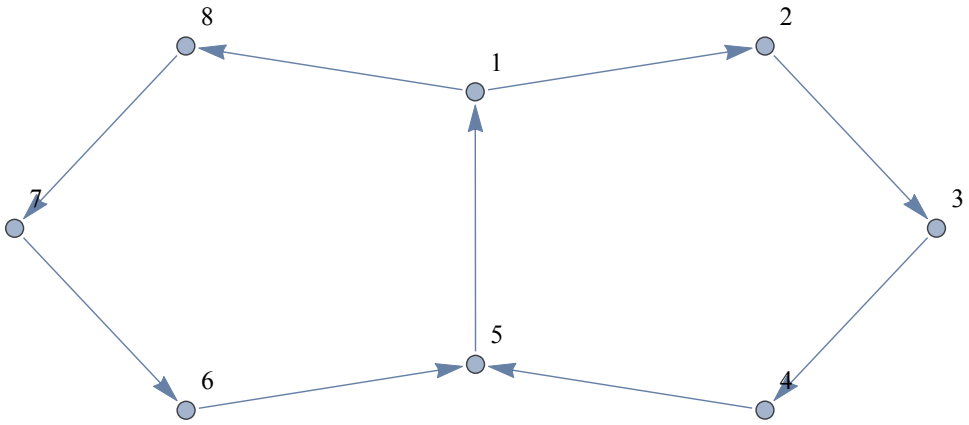
Izrek 3.1 *Vsak par vozlišč v krepko povezani komponenti tvori vsaj en cikel.*

Dokaz. Ta izrek izhaja iz definicije krepko povezanih komponent. Če sta vozlišči a in b v krepko povezani komponenti obstajata poti $a \to \cdots \to b$ in $b \to \cdots \to a$. Če ti dve poti združimo dobimo cikel $a \to \cdots \to b \to \cdots \to a$. □

Lahko se zgodi, da se povezava v ciklu pojavi večkrat (slika 3.1). V primeru, da se ponovi k-krat jo lahko brez vpliva na semantiko razdelimo na k novih povezav katerih teže so $\frac{w}{k}$ kjer je w teža zamenjane povezave.

[1] NIF je tehnologija v Erlangu, ki omogoča klice funkcij napisanih v C/C++.

To pomeni, da en dolg nadomestimo z več manjšimi, kar ne spremeni višine dolga, prilagodi pa graf tako, da lahko najdemo cikel. Tako vsakič ko cikel naleti na zamenjano povezavo uporabimo eno od novih povezav. Iz tega sledi, da lahko pri obdelavi cikla težo za namene iskanja najmanjše povezave nadomestimo z $\frac{w}{k}$ ob zmanjševanju povezav pa na tej povezavi zmanjšujemo za $k \cdot min$.



Slika 3.1: V ciklu med vozliščema 3 in 7 se povezava $5 \rightarrow 1$ ponovi dvakrat.

Cikel bi sicer lahko našli tudi brez SCC, vendar SCC ponujajo tudi dobro metodo delitve dela med delavce. Algoritem za iskanje krepko povezanih komponent deluje rekurzivno, zato lahko različne veje rekurzije izvajamo popolnoma neodvisno.

Izrek 3.2 *Odstranitev povezave znotraj krepko povezane komponente ne vpliva na prisotnost ali odsotnost ciklov zunaj krepko povezane komponente.*

Dokaz. Očitno je, da odstranjevanje povezav ne more ustvariti novega cikla. Dokažimo še, da ne more uničiti cikla, ki vsebuje vozlišča izven krepko povezane komponente.

Imamo graf $G\,(V, E)$, vozlišča $\{v_1, v_2, v_3\} \subset V$, povezavo $e_1 = (v_1, v_2) \in E$ in krepko povezano komponento $SCC \subset G$. Velja $\{v_1, v_2\} \subset SCC$ in $v_3 \notin SCC$.

Če bi obstajal tak cikel, da vsebuje povezavo e_1 in gre skozi v_3, potem bi obstajali tudi poti $v_1 \dots v_3$ in $v_3 \dots v_1$ in bi v_3 ustrezal definiciji vozlišča v SCC. Torej ne more obstajati tako vozlišče, ki ni v SCC ampak gre skozi njega cikel, ki vsebuje vozlišča, ki so del SCC. Obratno ne more obstajati cikel, ki vsebuje tako vozlišča, ki so v SCC, kot tista, ki niso. Torej lahko odstranimo povezave znotraj SCC, ne da bi vplivali na cikle zunaj SCC. □

Kot smo videli SCC in $\overline{SCC}$ lahko obdelujemo ločeno brez potrebe po usklajevanju. Ko izberemo vozlišče $v_p \in V$ lahko graf razdelimo na tri podmnožice: $\mathcal{F}_p$, $\mathcal{B}_p$ in $\mathcal{R}_p$. Množica $\mathcal{F}_p \subseteq V$ je množica vseh vozlišč v za katere obstaja pot $v_p \dots v$. Množica $\mathcal{B}$ pa je množica vseh vozlišč za katera obstaja obratna pot, torej $v \dots v_p$. Presek teh dveh množic je množica $\mathcal{SCC}$, torej krepko povezana komponenta, ki vsebuje v_p. Vozlišča, ki niso v nobeni od teh množic spadajo v množico $\mathcal{R}_p$.

Izrek 3.3 *Krepko povezane komponente tvorijo usmerjen acikličen graf.*

Dokaz. Kot smo prej dokazali je vsak cikel znotraj ene krepko povezane komponente. Če potem vsa vozlišča znotraj vsake komponente združimo v eno vozlišče, smo iz grafa odstranili vse cikle, saj so po izreku 3.2 cikli le znotraj krepko povezanih komponent in ne med njimi. Dobili smo usmerjen acikličen graf, kjer vsaka točka predstavlja eno krepko povezano komponento. □

Ko v SCC najdemo in obdelamo en cikel lahko SCC obdelujemo naprej kot popolnoma nov graf, ki vsebuje eno ali več krepko povezanih komponente. Enako lahko storimo z preostankom grafa, ki prav tako vsebuje tri ali več krepko povezanih komponent.

Izrek 3.4 *Naš celoten algoritem proizvede usmerjen acikličen graf.*

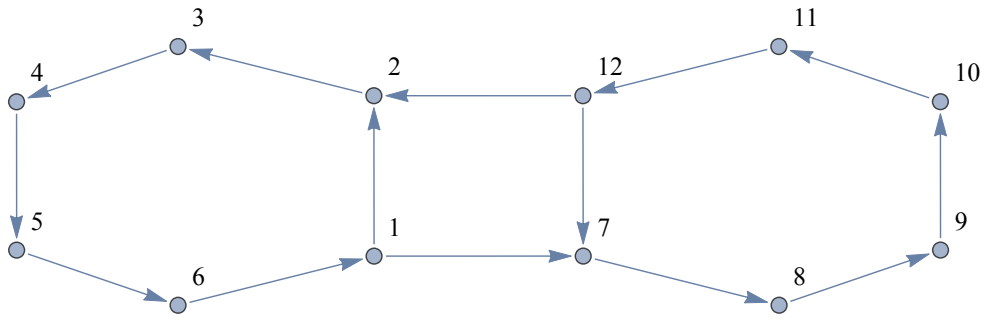
Dokaz. Krepko povezana komponenta velika eno vozlišče je že usmerjen acikličen graf.

Podgraf velik dve vozlišči ima eno ali dve krepko povezani komponenti in nič ali en cikel. Če sta komponenti dve je vsaka velika eno vozlišče, ki tvori

usmerjen acikličen graf in tako skupaj po izreku 3.3 tvorita usmerjen acikličen graf. Če je komponenta ena, obstaja en cikel. Če odstranimo eno povezavo na ciklu, kot to počne naš algoritem, je ciklov nič in imamo usmerjen acikličen graf.

Če iz krepko povezane komponente odstranimo cikel razpade na eno ali več krepko povezanih komponent velikosti eno ali več vozlišč. Komponente nato rekurzivno razpadajo dokler ne dosežejo velikosti enega vozlišča.

Če vsako komponento rekurzivno obdelamo dobimo nazaj usmerjene aciklične grafe, ki jih lahko nazaj združimo v usmerjen acikličen graf. □



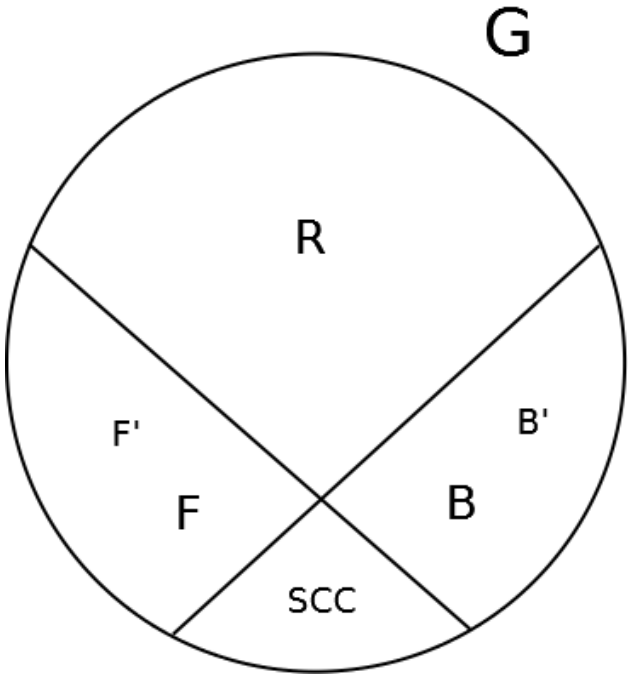
Slika 3.2: Če na tem grafu odstranimo povezavo $1 \to 7$ razpade na krepko povezani komponenti $\{1, 2, 3, 4, 5, 6\}$ in $\{7, 8, 9, 10, 11, 12\}$.

Lastnosti množic $\mathcal{F}'_p$, $\mathcal{B}'_p$ in $\mathcal{R}_p$

V nadaljevanju bomo potrebovali še množici $\mathcal{F}'_p = \mathcal{F}_p/\mathcal{SCC}$ in $\mathcal{B}'_p = \mathcal{B}_p/\mathcal{SCC}$, torej vozlišča iz $\mathcal{F}_p$ ali $\mathcal{B}_p$, ki niso v $\mathcal{SCC}$. Vozlišče v_p je vozlišče, iz katerega smo iskali $\mathcal{SCC}$, $\mathcal{SCC}'$ je hipotetična druga krepko povezana komponenta.

Izrek 3.5 *Krepko povezana komponenta v $\mathcal{F}'_p \cup \mathcal{B}'_p \cup \mathcal{R}_p$ je podmnožica natanko ene izmed $\mathcal{F}'_p$, $\mathcal{B}'_p$ in $\mathcal{R}_p$.*

Dokaz. Če velja $v_2 \in \mathcal{F}'$ in $v_3 \in \mathcal{B}'$ ter bi radi, da hkrati velja $\{v_2, v_3\} \subset \mathcal{SCC}'$ potem bi obstajala pot $v_2 \dots v_3$. Po definicijo množice $\mathcal{F}'_p$ bi obsta-



Slika 3.3: Vennov diagram množic.

jala tudi pot $v_p \dots v_2$. Obstajali bi torej poti $v_p \dots v_2 \dots v_3$ in, po definiciji množice B, $v_3 \dots v_p$. Torej bi vozlišči morali spadati v SCC.

Če velja $v_2 \in \mathcal{F}_p$ in $v_3 \in \mathcal{R}_p$ ter bi radi, da velja $\{v_2, v_3\} \subset \mathcal{SCC}'$ potem obstaja pot $v_p \dots v_2 \dots v_3$ in bi vozlišče v_3 spadalo v $\mathcal{F}_p$. Simetričen dokaz velja za $\mathcal{B}_p$ in $\mathcal{R}_p$.

Obstoj opisnih vozlišč je torej protisloven, kar pomeni, da krepko povezana komponenta, ki sega čez več kot eno od opisanih množic ne more obstajati. Za primere, ki vključujejo $\mathcal{SCC}'$ pa smo že prej dokazali v izreku 3.2, da niso mogoči. □

Izrek 3.6 *Vsaka od množic $\mathcal{F}'_p$, $\mathcal{B}'_p$ in $\mathcal{R}_p$ vsebuje vsaj eno krepko povezano komponento ali pa je množica prazna.*

Dokaz. Vsak graf vsebuje vsaj eno krepko povezano komponento, saj že eno samo vozlišče tvori krepko povezano komponento, in nobena krepko povezana komponenta se ne razteza čez več kot eno množico. □

Te lastnosti nato rekurzivno držijo za krepko povezane komponente znotraj teh množic.

Vse to lahko uporabimo v našem algoritmu. Ko bomo izračunali množice jih lahko v nadaljnjo obdelavo pošljemo različnim računalnikom, saj za njihovo obdelavo ne potrebujemo podatkov o vozliščih izven posamezne množice in tako predstavljajo enoto dela.

3.1.2 Iskanje krepko povezanih komponent

Za računanje krepko povezanih komponent bomo uporabljali model PRAM CREW. Ta načeloma omogoča uporabo poljubno število procesorjev, omejuje pa ga vrsta problema. V tem primeru lahko uporabimo najmanj 1 in največ $|E|$ procesorjev.

Uporabili bomo algoritem FW-BW iz Hong et al. [12]. Ta vzame graf $G(V, E)$ in si izbere poljubno vozlišče v_p. Nato poišče množici F in B, naredi njun presek in poišče množico R ter izdela F' in B'.

Poti najdemo z iskanjem v širino, ki ga začnemo v v_p. Zato, da bomo kasneje lažje našli cikel si v tej fazi za vsako vozlišče zapišemo, po kateri povezavi smo prišli do njega, da imamo na koncu množico poti od vsakega vozlišča do v_p in/ali obratno.

Za iskanje v širino lahko uporabimo vzporeden algoritem.

Algoritem 4 po plasteh obdeluje graf. Trenutno plast hrani v spremenljivki C. Množico C lahko brez težav obdeluje vzporedno, tako da za vsako vozlišče poišče vse sosede, preveri kateri od teh še niso bili obiskani in te da v množico N. Ko je množica C obdelana jo zamenja z množico N in ponovi zanko.

Izrek 3.7 *Algoritem 4 opravi $O(|V| + |E|)$ dela, kar je optimalno.*

Dokaz. Algoritem obišče vsako vozlišče največ enkrat, saj obiskana vozlišča označi in jih v primeru, da ponovno naleti na njih ne dodaja v množico za nadaljnje obdelovanje.

Algoritem 4: Vzporedni BFS algoritem povzet po Hong et al. [11].

```
C = ∅ ;
N = {v_p x};
From[] = 0;
repeat
    C = N;
    N = {};
    foreach v ∈ C pardo
        foreach v_n : (v, v_n) ∈ E pardo
            if From[v_n] = 0 then
                /* Označi vozlišče v_n in nastavi podatke o
                   poti.                                    */
                From[v_n] = v;
                N = N ∪ {v_n}
            end
        end
    end
until N = {};
```

Algoritem obišče vsako povezavo največ enkrat, saj v vsakem vozlišču pregleda vse povezave, ki gredo iz vozlišča. Ker vsako vozlišče obišče največ enkrat je nemogoče, da bi povezavo obiskal več kot enkrat.

Vzporeden algoritem se šteje za optimalen, če je količina dela, ki ga procesorji opravijo skupaj asimptotično enaka časovni zahtevnosti zaporednega algoritma [13].

Časovna zahtevnost zaporednega algoritma za preiskovanje v širino je $O(|V| + |E|)$ [6] torej je algoritem 4 optimalen. □

Sedaj imamo štiri množice (SCC, F', B' in R), ki so, kot smo dokazali, med seboj neodvisne. V podgrafu SCC smo tudi odstranili eno povezavo, kar pomeni, da o njegovi povezanosti ne vemo ničesar. Logično gledano sedaj rekurzivno v vsaki od njih poiščemo novo krepko povezano komponento in

tako dalje. Ker pa delo želimo porazdeliti med več računalnikov pošljemo te podgrafe glavnemu strežniku, ki opravi razporejanje.

Iz povednega sledi, da algoritem opravi za vsak cikel $O(|V| + |E|)$ dela. Graf z vozlišči, ki imajo izhodno stopnjo k ima $\Omega(k^2)$ ciklov, ki si ne delijo povezav, domneva pa se, da je ciklov vsaj $\binom{k+1}{2}$ [4]. Imenujmo število ciklov ζ. To pomeni, da ima poln graf $\zeta = \Omega\left(|V|^2\right)$ ciklov . Za naš algoritem to pomeni, da v najslabšem primeru opravi $O(\zeta \cdot (|V| + |E|))$ dela. Enako velja za algoritem 5, kar pomeni, da če algoritem 5 vzamemo za najboljši zaporedni algoritem lahko naš algoritem proglasimo za optimalen. Naši empirični testi so pokazali, da je v grafih, ki jim je algoritem namenjen ciklov $O(|V|)$, kar nam prinese kvadratično količino dela.

Algoritem 5: Preprost zaporeden algoritem.

```
repeat
    /* Poišči cikel.                    */
    C = DFS();
    /* Obdelaj cikel.                   */
    Obdelaj(C);
until V grafu ni več ciklov;
```

Izrek 3.8 *Časovna zahtevnost našega algoritma je v najslabšem primeru* $O\left((|V| + |E|)^2\right)$.

Dokaz. Če je graf sestavljen iz ene same poti na vsakem nivoju rekurzije potrebujemo $O(|V| + |E|)$ za pregled v širino.

Če za pivotno vozlišče vsakič vzamemo vozlišče na začetku ali koncu poti graf razpade na SCC velikosti ena in množico F' ali B', ostali množici pa ostaneta prazni, torej v vsaki iteraciji iz grafa odstranimo eno vozlišče. Rekurzij je torej $O(|V| + |E|)$ in pri vsaki porabimo $O(|V| + |E|)$ časa za DFS. □

3.1.3 Iskanje cikla

Ko smo našli množico SCC poljubni točki v njej tvorita vsaj en cikel. Pri iskanju v širino algoritem dopolnimo, da ob dodajanju vozlišč v množico za obdelavo za vsako dodano vozlišče zapiše še, da vanj pridemo iz trenutnega vozlišča. Tako dobimo poti od vseh vozlišč do vozlišča v_1 in poti od v_1 do vseh ostalih vozlišč v SCC.

Te poti torej uporabimo, da zgradimo cikel, ki vključuje pivotno vozlišče in poljubno drugo vozlišče tako, da se iz pivotnega vozlišča premaknemo po poljubni poti poljubno daleč znotraj SCC, nato pa uporabimo povratno pot, da zaključimo cikel. Časovna zahtevnost tega postopka je $O(|V| + |E|)$.

3.2 Implementacija iskanja SCC

Ta del programa bomo implementirali v C s pomočjo standarda OpenMP. Najbolj pomembno je narediti obdelovanje vozlišč na trenutnem nivoju vzporedno. Tu pride prav OpenMP konstrukt vzporedne `for` zanke. Z vzporedno zanko bomo obdelovali množico C, tukaj predstavljeno kot polje referenc na vozlišča.

Težava se pokaže, ko sestavljamo množico N, ki predstavlja naslednjo plast vozlišč. Kot predpostavlja PRAM CREW model bi v primeru, da bi vse niti hotele hkrati pisati vanjo prišlo do močne upočasnitve, saj bi morale čakati ena drugo, teh pisanj pa je veliko. K sreči gre za množico, torej vrstni red ni pomemben, zato lahko vsaka nit med delovanjem piše v svojo lokalno množico, na koncu zanke pa se sinhronizirajo in množice združijo [11]. Ker se množice hranijo v obliki povezanih seznamov jih združimo tako, da staknemo prve in zadnje člene. Tako dosežemo, da atomarni dostopi in sinhronizacija niso potrebni v vsaki iteraciji notranje zanke temveč le po koncu zunanje.

Prav tako ni potrebna sinhronizacija pri obdelovanju vozlišč, saj vse niti pišejo enake ali enakovredne podatke. Kadar vozlišča označujejo za obiskana vse niti pišejo isti podatek, ko pa vpisujejo podatke o poti pa ni pomembno katera nit napiše svoje vozlišče kot predhodnik ali naslednik, saj so si med

seboj enakovredni le končna pot je drugačna.

3.3 Implementacija porazdeljenega sistema

Za porazdeljen sistem potrebujemo glavni strežnik in delovne strežnike. Glavni strežnik pripravi podatke tako, da jih razdeli na enote dela in jih da v vrsto. Nato čaka, da mu delovni strežnik sporoči, da je pripravljen. Ko se ta prijavi, glavni strežnik iz vrste vzame enoto dela, v našem primeru podgraf, in jo pošlje delovnemu strežniku. Ta podatke obdela in rezultate, ti so pri nas podgrafi SCC, F, B in R, vrne glavnemu strežniku, ta jih da v vrsto za delo, delovni strežnik pa ponovno čaka na dodeljevanje dela. Lahko se zgodi, da med delom delovni strežnik odpove ali izgubi stik z glavnim strežnikom, zato si glavni strežnik ob razdeljevanju zapomni katero delo je dal kateremu delovnemu strežniku. V primeru, da zazna izpad delo uvrsti nazaj v vrsto.

Implementacija je izdelana s pomočjo Erlanga in ogrodja Erlang/OTP. Erlang je programski jezik zasnovan za porazdeljeno programiranje [8] in vsebuje programske konstrukte namenjene posebej medprocesni komunikaciji.

Uporabili bomo dve implementaciji `gen_server` strežnika iz ogrodja OTP. Ena bo skrbela za razporejanje dela, druga pa bo delo sprejemala in ga s pomočjo prej opisane vzporedne funkcije obdelala. Ko bo graf razdelila na opisane štiri množice jih bo poslala nazaj glavnemu strežniku, ta pa jih bo ponovno razdelil med delovne strežnike.

Narava povezav med Erlang strežniki omejuje velikost gruče na okoli 50-150, odvisno od sposobnosti mreže, vendar, kot bomo videli kasneje, to ni problematično.

3.3.1 Glavni strežnik

Glavni strežnik skrbi za razporejanje dela in sledenju razpoložljivosti delovnih strežnikov. Del kode, ki opisuje komunikacijo je v dodatku A, diagram, ki prikazuje primer delovanja po opisanem protokolu pa je na sliki 3.4. Ob

postavitvi naloži graf in ga pripravi v obliki, ki je primerna za pošiljanje delovnim strežnikom.

Nato čaka na zahteve delovnih strežnikov za pridružitev. Kot stanje hrani tri sezname: vrsto prostih delovnih strežnikov QN, vrsto enot dela QW in preslikavo (ang. *map*) med razdeljenimi enotami dela in delovnimi strežniki MW, kjer je označeno, kateri strežnik ima dodeljeno katero enoto dela. Pomen slednjega si bomo ogledali kasneje.

Delovni strežnik se pridruži tako, da pošlje zahtevo `join`, ki vsebuje njegovo ime. Ko glavni strežnik to zahtevo sprejme, samemu sebi pošlje zahtevo `add_node` za dodajanje delovnega strežnika v vrsto.

Ko delovni strežnik opravi delo pošlje podatke glavnemu strežniku v obliki `return_cycle` zahteve. Ta vzame iz zahteve množice F', B' in R in jih samemu sebi pošlje kot asinhrono zahtevo `queue_work` za dodajanje dela v vrsto. Še prej pa si pošlje zahtevo `eliminate_cycle` s ciklom in množico SCC' kjer še v svoji lokalni evidenci obdela cikel in SCC' pošlje v vrsto dela. Na koncu še doda strežnik, ki je poslal podatke nazaj v vrsto prostih delovnih strežnikov z zahtevo `add_node`.

Zahteva `add_node` se obdela tako, da najprej pogleda, če je v vrsti za delo na voljo prosta enota dela. Če je ta prisotna pokliče notranjo funkcijo za pošiljanje dela delovnemu strežniku. Če dela ni potem delovni strežnik doda na seznam čakajočih strežnikov. Podobno deluje tudi zahteva `queue_work`, ki gleda za proste strežnike in v primeru da teh ni doda delo na seznam prostih enot dela.

Funkcija za dodeljevanje dela delovnemu strežniku pošlje zahtevo `find` v kateri je graf. Hkrati pa graf in strežnik doda v preslikavo.

Glavni strežnik tudi sledi dogodkom porazdeljenega sistema. Če delovni strežnik postane neodziven ali samo preneha z delovanjem je glavni strežnik o tem obveščen. V preslikavi poišče delo, ki je bilo strežniku dodeljeno in ga doda nazaj v vrsto, strežnik pa odstrani iz seznama prostih delovnih strežnikov. Tako zagotovi, da ob morebitnem izpadu delovnih strežnikov delo poteka naprej neprekinjeno in brez napak.

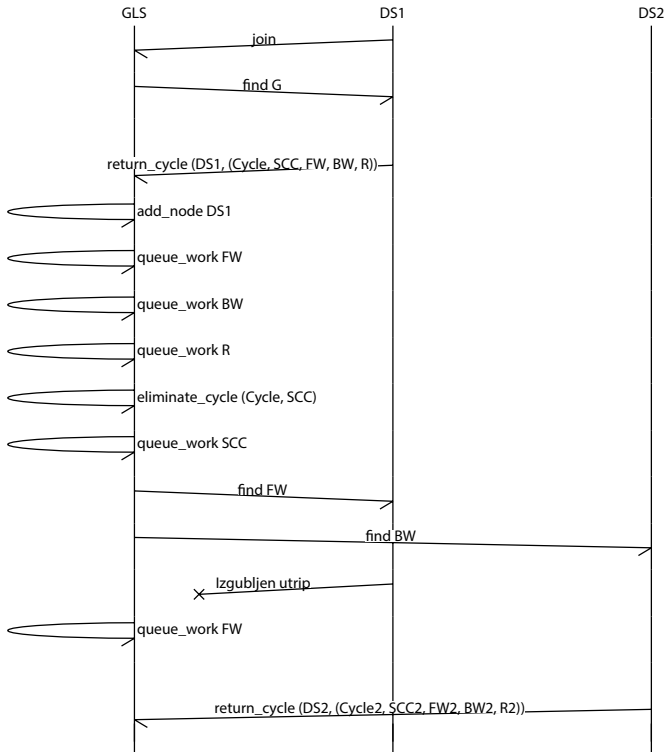
3.3.2 Delovni strežnik

Naloga delovnega strežnika je, da prevzame graf[2], v njem poišče množice SCC, F', B' in R, nato v SCC poišče en cikel in obdela ter ob tem odstrani eno povezavo da dobi SCC'. Na koncu vse štiri množice (SCC', F', B', R) pošlje glavnemu strežniku. Koda, ki opisuje komunikacijo je v dodatku B.

Ko se strežnik zbudi pošlje glavnemu strežniku asinhrono zahtevo, v kateri pove, da se želi pridružiti delu.

Strežnik kot asinhrono zahtevo `find` dobi graf kot seznam povezav oblike `#edge{source = A, destination = B, weight = W}`. Tega kar naravnost uporabi za klic funkcije `find_cycle`, ki je kot je bilo prej opisano implementirana s pomočjo NIF knjižnice. Ta vrne četverico oblike `(Cikel, SCC, FW, BW, R)`. Ti podatki se skupaj z imenom strežnika pošljejo kot asinhrona zahteva `return_cycle` nazaj glavnemu strežniku.

[2]Lahko gre tudi za podgraf, a kot smo prej pokazali za delovanje razlike ni.



Slika 3.4: Diagram protokola, ki se uporablja za razdeljevanje dela. GS je glavni strežnik, DS1 in DS2 pa sta delovna strežnika.

Poglavje 4

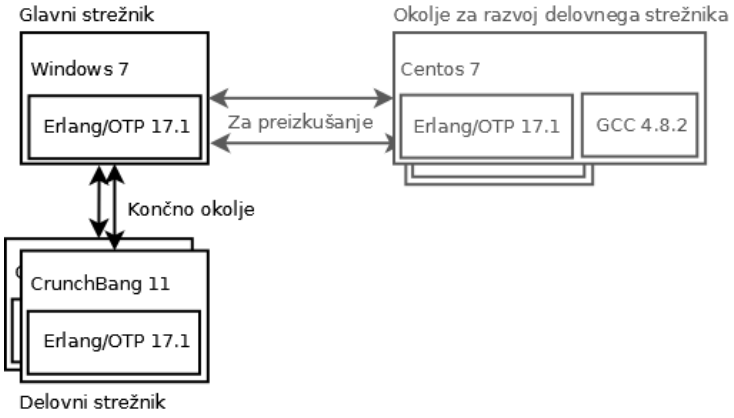
Empirična analiza rezultatov

4.1 Okolje

Za implementacijo porazdeljenega dela algoritma je bilo uporabljeno ogrodje Erlang/OTP 17.1 na operacijskem sistemu Windows 7 za glavni strežnik, CentOS 7 za razvoj delovnega strežnika in CrunchBang 11 za delovne strežnike[1]. Sistema, ki uporabljata Linux sta bila izbrana zaradi lažjega prevajanja C kode.

Kode napisana v C je bila prevedena s prevajalnikom GCC 4.8.2 s podporo za OpenMP 3.1 v CentOS 7.

[1]CentOS in CrunchBang sta različici Linuxa.



Slika 4.1: Topologija testnega okolja. Puščice predstavljajo prenose podatkov med delovnim in glavnim strežnikom med delovanjem.

V glavnem strežniku uporabljen procesor Intel Core i5-560M 2.6 GHz, delovni strežnik pa je uporabljal virtualni računalnik na osnovi VirtualBoxa na procesorju Intel Core i7-2600K 3.4GHz. Virtualni računalnik je uporabljal štiri jedra in 2GB pomnilnika brez navideznega pomnilnika. Strežnika sta bila povezana z 100 Mb/s Ethernet povezavo. Za preizkušanje porazdeljenega delovanja je bil na glavnem strežniku še en virtualni delovni strežnik.

4.2 Grafi

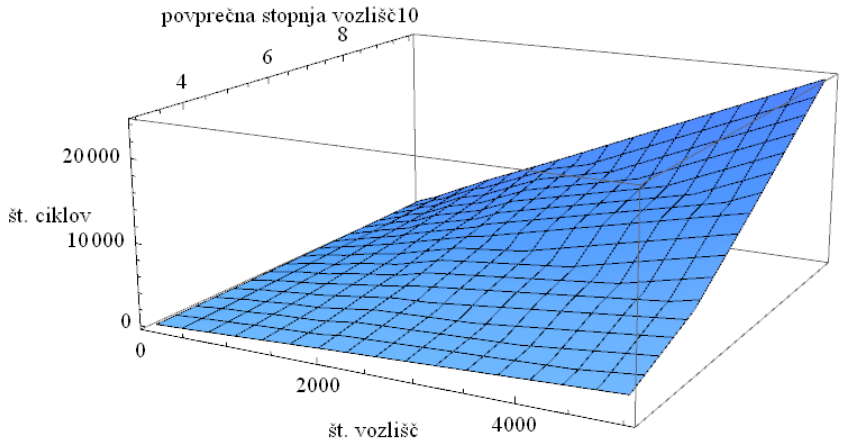
Grafi so naključno generirani grafi majhnega sveta, narejeni s pomočjo ogrodja SNAP. Gre za grafe po modelu Watts–Strogatz, z izhodnimi stopnjami 3, 5, 7 in 10 (tabela 4.1) [14].

Graf dobljenih rezultatov (slika 4.2) pokaže, da je relacija med številom ciklov in velikostjo grafa oblike $a\,|V| + b\,|V|\,D_{out}$ in ker velja $|V|\,D_{out} = |E|$, to pomeni, da je ciklov $O\,(|V| + |E|)$. Ker pa je $|E| = O\,(|V|)$ velja, da je ciklov $O\,(|V|)$.

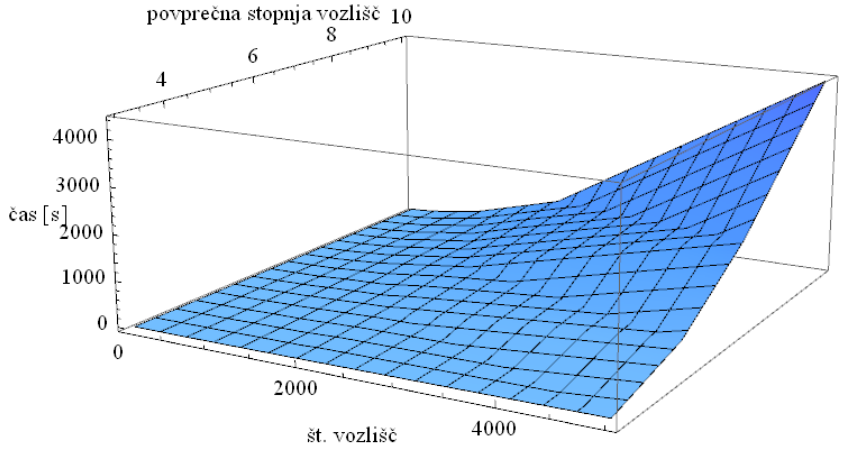
Meritve časa kažejo na polinomsko rast (slika 4.2b), kar ustreza pričakovanjem.

št. vozlišč	povprečna stopnja vozlišč	začetno št. povezav	št. odstranjenih ciklov	čas (s)
100	3	299	51	0.11
100	5	497	106	0.67
100	7	686	243	1.56
100	10	966	457	3.54
300	3	897	143	0.55
300	5	1494	406	3.99
300	7	2087	791	9.72
300	10	2973	1278	20.79
500	3	1499	233	2.54
500	5	2496	695	8.84
500	7	3488	1222	20.06
500	10	4973	2305	60.06
1000	3	2998	485	4.56
1000	5	4995	1224	32.23
1000	7	6987	2435	84.68
1000	10	9976	4840	204.78
2000	3	5998	1008	18.92
2000	5	9989	2695	106.55
2000	7	13987	5234	325.00
2000	10	19980	9526	748.51
5000	3	14998	2431	71.01
5000	5	24993	6283	602.21
5000	7	34989	13237	1835.27
5000	10	49978	24135	4390.51

Tabela 4.1: Testni grafi.



(a) Graf števila ciklov



(b) Graf izmerjenih časov

Slika 4.2: Graf števila ciklov v odvisnosti od števila vozlišč in izhodno stopnjo vozlišč in graf izmerjenih časov izvajanja.

4.3 Algoritem

Algoritem, ki smo ga uporabili se da v grobem opisati v dveh fazah. V prvi fazi v grafu poiščemo eno krepko povezano komponento in ji odstranimo en cikel. Ta faza se odvija na posameznem delovnem strežniku (glej sliko 4.1) vzporedno. V drugi fazi postopek ločeno ponovimo na krepko povezani komponenti in preostanku grafa. V tej fazi delo razporedimo med

delovne strežnike v porazdeljenem sistemu (slika 4.1).

V praksi se izkaže, da na grafu majhnega sveta, grafu kjer je razdalja med pari vozlišč majhna v primerjavi s številom vozlišč, porazdeljeni del algoritma ne more popolnoma izkoristiti svoje porazdeljene narave, torej iz vsake obdelane enote dela nastane le ena nova enota. Dobra povezanost grafov majhnega sveta pomeni, da grafe navadno sestavlja ena krepko povezana komponenta velikosti $O(|V|)$ in veliko relativno majhnih komponent [12]. To pomeni, da, ko so manjše komponente obdelane, kar je sorazmerno hitro, ostane le večja komponenta. Na več komponent ta večja komponenta sicer razpade proti koncu izvajanja, ko je odstranjena večina ciklov, vendar večino časa delo opravlja lahko le en delovni strežnik.

Možno je, da bi manjše komponente pošiljali v obdelavo šibkejšim računalnikom, vendar je vrednost takih strategij močno odvisna od hitrosti povezav. Te bi pri majhnih enotah dela že z minimalnimi zamiki lahko upočasnile delo. Je pa taka strategija lahko boljša z vidika porabe moči, saj lahko dodatne strežnike vključimo le na začetku in koncu dela, vmes pa so izključeni. To je sicer dobro, ker Erlang ne podpira velikih gruč, vendar pomeni, da morajo biti računalniki v gruči močnejši.

4.4 Implementacija

Implementacija se je izkazala za problematično. Podatki se prenašajo v obliki seznama povezav grafa. Pretvorbe podatkov iz oblike, ki jo uporablja in razpošilja Erlang, v obliko, ki jo uporablja C so relativno počasne in se jih ne da izvajati vzporedno. Erlang uporablja način upravljanja s seznami, ki ga vidimo v funkcijskih jezikih, torej delitev na glavo in rep. To pomeni, da seznama ni mogoče obdelovati drugače kot zaporedno. Za vzporedno delo je najboljša oblika polje, kar pomeni, da je potrebno povezan seznam prepisati v polje.

Ena od možnih rešitev bi bilo razpošiljanje podatkov v binarni obliki, vendar bi s tem izgubili mnogo prednosti, ki jih nudi visokonivojsko progra-

miranje, kot na primer neodvisnost od binarne predstavitve, velikost implementacije pa bi močno narasla.

Poglavje 5

Zaključek

Implementacija se je izkazala za zelo problematično, vendar smo uspeli pridobiti empirične podatke o naravi problema. Na podlagi teh lahko rešitev izboljšamo in prilagodimo podatkom iz resničnega sveta.

5.1 Izboljšave

Algoritem kot celota ima kubično zahtevnost, kar navadno pomeni, da obstaja možnost izboljšave.

5.1.1 Porazdeljeni del

Porazdeljeni del algoritma se je izkazal za precej požrešnega, saj v primerjavi z delom implementiranim v C predstavlja precejšen delež časa. Tukaj sta problematična pasovna širina in zakasnitev mrežnih povezav ter pretvorbe podatkov. Še posebno pretvorbe porabijo veliko več časa, kot bi za ta del algoritma pričakovali. V nalogi smo analizirali oziroma opisali BOINC, MPI in Erlang ter prišli do sledečih zaključkov.

BOINC

BOINC je namenjen problemom, kjer prenos podatkov terja sorazmerno malo sredstev v primerjavi z obdelavo le-teh, zato za izvajanje tega algoritma brez

velikih sprememb ni primeren.

MPI

Ker je MPI namenjen implementaciji v C z njegovo uporabo ne bi bilo izgub zaradi pretvorb. Prav tako bi bil prenos vsaj tako hiter kot pri Erlangu, izgubili pa bi stabilnost in vgrajene mehanizme za porazdeljeno delovanje, ki jih ima Erlang. Uporaba MPIja bi zelo verjetno močno izboljšal hitrost, brez potrebe po izboljšavah, ki zahtevajo neobičajno rabo jezika. Na novo pa bi bilo potrebno implementirati razporejevalnik dela, saj je MPI samo orodje za medprocesno komunikacijo in takšnih funkcionalnosti še nima.

Erlang

Za Erlang smo se odločili, ker nas je zanimalo, če je implementacija takih sistemov z njegovo pomočjo mogoča in smiselna.

Čisti Erlang se je izkazal za neprimernega, kar pomeni da bi za nadaljnje optimizacije morali uporabiti pristope na nižjem nivoju, kot je na primer prenos podatkov v posebej za ta problem narejeni binarni obliki. V tem primeru pa bi se verjetno za bolj učinkovitega izkazal MPI, ki v vsakem primeru deluje na nižjem nivoju.

5.1.2 Vzporedni del

Vzporedni del algoritma je bil v tej implementaciji v primerjavi z razdeljevanjem izredno hiter. Izboljšave so še vedno možne. Ena možnost je implementacija večjega dela algoritma iz Hong et al. [12], ki odreže določene trivialne dele grafa in jih obdela hitreje. Prav tako je možna izboljšava z uporabo splošnonamenskih enot v grafičnih karticah, vendar ni smiselna, dokler se močno ne izboljša porazdeljeni del algoritma.

Dodatki

Dodatek A

Del kode glavnega strežnika

Listing A.1: Del kode glavnega strežnika.

```
handle_cast({join, Node}, State) ->
  gen_server:cast({global, diploma_master_server},
   {add_node, Node}),
  {noreply, State};

handle_cast({return_cycle,
 {Node, {Cycle, Scc, Fw, Bw, Rest}}}, State) ->
  NewState = State#state{assigned_work = maps:remove(
        Node, State#state.assigned_work)},
  gen_server:cast({global, diploma_master_server},
   {add_node, Node}),
  gen_server:cast({global, diploma_master_server},
   {eliminate_cycle, Cycle, Scc}),
  gen_server:cast({global, diploma_master_server},
   {queue_work, Fw}),
  gen_server:cast({global, diploma_master_server},
   {queue_work, Bw}),
  gen_server:cast({global, diploma_master_server},
   {queue_work, Rest}),
```

```
  {noreply, NewState};

handle_cast({eliminate_cycle, Cycle, Scc}, State) ->
  eliminate_cycle(Cycle),
  gen_server:cast({global, diploma_master_server},
   {queue_work, Scc}),
  {noreply, State};

handle_cast({queue_work, GraphNodes}, State)
 when length(GraphNodes) > 1 ->
  case queue:out(State#state.free_nodes) of
    {{value, Node}, NewNodeQueue} ->
      NewState = assign_work(Node, GraphNodes, State),
      {noreply, NewState#state{
        free_nodes = NewNodeQueue}};
    {empty, _Queue} ->
      NewWorkQueue = queue:in(
        GraphNodes, State#state.work_queue),
      {noreply, State#state{work_queue = NewWorkQueue}}
  end;

handle_cast({add_node, Node}, State) ->
  case queue:out(State#state.work_queue) of
    {{value, GraphNodes}, NewWorkQueue} ->
      NewState = assign_work(Node, GraphNodes, State),
      {noreply, NewState#state{
        work_queue = NewWorkQueue}};
    {empty, _Queue} ->
      NewNodeQueue = queue:in(
      Node, State#state.free_nodes),
      {noreply, State#state{free_nodes = NewNodeQueue}}
```

```
  end;

handle_cast(_Request, State) ->
  {noreply, State}.

handle_info({nodedown, Node}, State) ->
  Map = case maps:get(
        Node,
        State#state.assigned_work,
        none_assigned) of
          none_assigned ->
            State#state.assigned_work;
          GraphNodes ->
            gen_server:cast(
             {global, diploma_master_server},
             {queue_work, GraphNodes}),
            maps:remove(
             Node, State#state.assigned_work)
        end,
  NewQueue = queue:filter(
        fun(Item) ->
                Item =/= Node end,
                State#state.free_nodes),
  {noreply, State#state{
        free_nodes = NewQueue, assigned_work = Map}};
```

Dodatek B

Del kode delovnega strežnika

Listing B.1: Del kode delovnega strežnika.

```
handle_cast({find, SubgraphEdges}, State) ->
  % Funkcija find_cycle je implementirana
  % kot NIF funkcija.
  Result = find_cycle(SubgraphEdges),
  gen_server:cast({global, diploma_master_server},
   {return_cycle, {node(), Result}}),
  {noreply, State};

handle_cast(_Request, _State) ->
  erlang:error(not_implemented).
```

Literatura

[1] The BOINC application programming interface. https://boinc.berkeley.edu/trac/wiki/BasicApi. Dostopano 2014-08-28.

[2] Tarjan's strongly connected components algorithm. https://en.wikipedia.org/wiki/Tarjan Dostopano 2014-09-11.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data structures and algorithms*. Addison-Wesley, Reading, MA, 1983.

[4] Noga Alon, Colin McDiarmid, and Michael Molloy. Edge-disjoint cycles in regular directed graphs. *Journal of Graph Theory*, 22(3):231–237, 1996.

[5] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[7] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] Ericsson. *Erlang/OTP System Documentation 6.1*. Ericsson AB, 2014.

[9] Message Passing Interface Forum. *MPI: a message passing interface standard*. High Performance Computing Centre, 2012.

[10] Khronos OpenCL Working Group et al. The OpenCL specification. *version*, 1(29):8, 2008.

[11] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88. IEEE, 2011.

[12] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 92. ACM, 2013.

[13] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[14] Matija Rezar. Vhodni podatki – grafi. `http://lusy.fri.uni-lj.si/sites/lusy.fri.uni-lj.si/files/publications/mrezar_graphs_20140912.zip`, september 2014.

[15] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[16] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[17] Victor V. Zhirnov, Ralph K. Cavin, James A. Hutchby, and George I. Bourianoff. Limits to binary logic switch scaling-a gedanken model. *Proceedings of the IEEE*, 91(11):1934–1939, 2003.